Gof Design Patterns

What is the training all about?

- Common problems
- Common solutions
- Trade offs

Consider a common problem

My web site has the same stuff at the top and bottom of each page and I don't want to duplicate that in every page!

Solution
Put the common stuff in separate files (header.html and footer.html) and include them on every page.
Trade Offs
- No duplication of header and footer code now (+)
- Still have boilerplate text (includes etc) in every page(-)
- If some pages need a different header / footer, you either make new header / footer files for that page or add conditional logic to the header / footer files (-)

There are other possible solutions, that would also have different trade offs.

These are simple designs which are used day to day by a developer

3

Patterns provide a common vocabulary

"An important part of patterns is trying to build a common vocabulary, so you can say that this class is a Remote Facade and other designers will know what you mean."

-- Martin Fowler

4

What is a Pattern

Patterns have four parts:

- Name - the common vocabulary
- Problem - "forces" that determine when the pattern is applicable
- Solution - a **template for solving the problem**
- Consequences - "pros **and cons**"

5

Design patterns *capture solutions that have evolved over time as developers strive for greater flexibility in their software. Whereas class libraries are reusable source code, and components are reusable packaged objects, patterns are generic, reusable design descriptions that are customized to solve a specific problem. The study of design patterns provides a common vocabulary for communication and documentation, and it provides a framework for evolution and improvement of existing patterns.*

6

OOPS coding and design principles

- Prefer delegation over inheritance
- Program to an interface not to an implementation

9

SOLID PRINCIPLES

Single Responsibility Principle

10

```
public class UserService
{
   public void Register(str
   {
      if (!ValidateEmail(em
         throw new Validati
         var user = new Use

         SendEmail(new Mail
   }
   public virtual bool Vali
   {
      return email.Contains(
   }
   public bool SendEmail(Ma
   {
      _smtpClient.Send(messa
   }
}
```

```
public class UserService
{
   EmailService _emailService;
   DbContext _dbContext;
   public UserService(EmailService aEmailService, DbContext aDbContext)
   {
      _emailService = aEmailService;
      _dbContext = aDbContext;
   }
   public void Register(string email, string password)
   {
      if (!_emailService.ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);
         _dbContext.Save(user);

emailService.SendEmail(new MailMessage("myname@mydomain.com", email) {Subject="Hi. How are y

      }
   }
   public class EmailService
   {
      SmtpClient _smtpClient;
   public EmailService(SmtpClient aSmtpClient)
   {
      _smtpClient = aSmtpClient;
   }
   public bool virtual ValidateEmail(string email)
   {
      return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
      _smtpClient.Send(message);
   }
}
```

Open Closed Principle

11

```
01. public abstract class Shape
02. {
03.    public abstract double Area();
04. }
```

```
01. public class Rectangle: Shape
02. {
03.    public double Height {get;set;}
04.    public double Width {get;set;}
05.    public override double Area()
06.    {
07.       return Height * Width;
08.    }
09. }
10. public class Circle: Shape
11. {
12.    public double Radius {get;set;}
13.    public override double Area()
14.    {
15.       return Radius * Radus * Math.PI;
16.    }
17. }
```

```
01. public class AreaCalculator
02. {
03.    public double TotalArea(Shape[] arrShapes)
04.    {
05.       double area=0;
06.       foreach(var objShape in arrShapes)
07.       {
08.          area += objShape.Area();
09.       }
10.       return area;
11.    }
12. }
```

```
28.       return area;
29.    }
30. }
```

Liskov Substitution Principle

```
public class Sc
{
```

```
public class SqlFile
{
   public string LoadText()
```

```
public interface IReadableSqlFile
{
   string LoadText();
}
public interface IWritableSqlFile
{
   void SaveText();
}
```

```
public class SqlFile: IWritableSqlFile,IReadableSqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
   public void SaveText()
   {
      /* Code to save text into sql file */
   }
}
```

```
public class ReadOnlySqlFile: IReadableSqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
}
```

```
public class SqlFileManager
{
   public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in aLstReadableFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
   {
   foreach(var objFile in aLstWritableFiles)
   {
      objFile.SaveText();
   }
   }
}
```

```
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles()
   {
      foreach(var objFile in lstSqlFiles)
      {
         objFile.SaveText();
      }
   }
}
```

```
lling it's
```

```
      if(! objFile is ReadOnlySqlFile)
         objFile.SaveText();
   }
}
```

Interface Segregation Principle

13

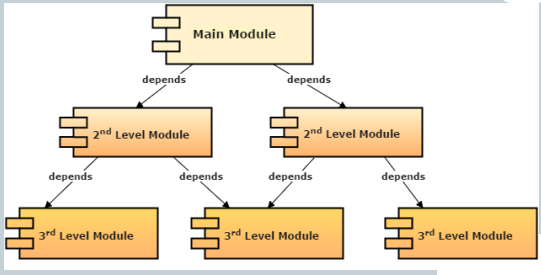
```
public Interface ILead
{
    void CreateSubTask();
    void AssginTask();
    void WorkOnTask();
}
public class TeamLead : ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task
    }
    public void WorkOnTask()
    {
        //Code to implement perform assigned task.
    }
}

public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task.
    }
    public void WorkOnTask()
    {
        throw new Exception("Manager can't work on Task");
    }
}
```

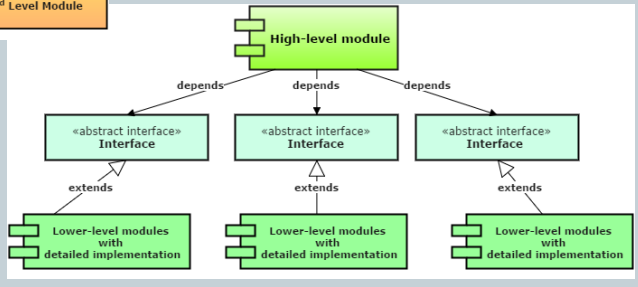
```
public interface IProgrammer
{
    void WorkOnTask();
}

public interface ILead
{
    void AssignTask();
    void CreateSubTask();
}

public class Programmer: IProgrammer
{
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
    //Code to create a sub taks from a task.
    }
}
```

```
public class TeamLead: IProgrammer, ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
        //Code to create a sub task from a task.
    }
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
```

Dependency Inversion Principle

14





SOLID Principles

- Single responsibility principle
  - a class should have only a single responsibility (i.e. only changes to one part of the software's specification should be able to affect the specification of the class).
- Open/closed principle
  - "software entities … should be open for extension, but closed for modification."
- Liskov substitution principle
  - "Derived types must be completely substitutable for their base types".
- Interface segregation principle
  - "many client-specific interfaces are better than one general-purpose interface."
- Dependency inversion principle
  - one should "depend upon abstractions, [not] concretions."
  - Abstractions should not depend upon details. Details should depend upon abstractions.

Other coding principles

16

- Use code naming standards
- Restrict method length
- Avoid unnecessary logging
- Keep code neat. Avoid comments. Code should be self explanatory
- Delete outdated code and comments
- DRY – Do not repeat yourself

Introduction to UML

UML Class Diagram

Introduction to UML Class Diagram

A class diagram consists of a group of classes reflecting important entities of the business domain of the system being modeled and the relationships between the classes and interfaces.

It is a pictorial representation of the detailed system design and provides a static view of the system. The structure of a system is represented using class diagrams.

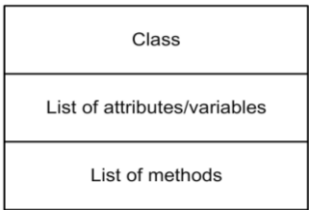
The elements that comprise a class diagram are as follows:

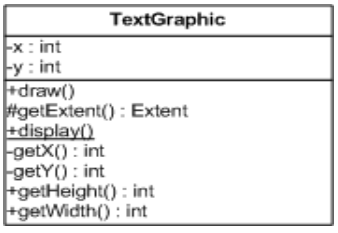
- Classes
- Interfaces
- Packages
- Relationships/links

UML Class Diagram

Elements of a Class Diagram

- Class: A class represents an entity of a given system and provides an encapsulated implementation of a certain functionality of a given entity.





Notes and constraints can be added to a list of attributes. Notes contain additional information that can be used as a reference while developing a system, while constraints are the business rules that a class must follow and are included as text in curly braces.

19

Modeling the Static Structure

Links and Relationships

A relationship is a logical connection found between class and object.

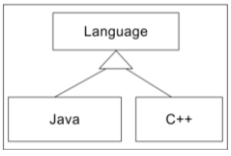
A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, the attributes and operations (or methods) of the classes, and the relationships among the classes.

20

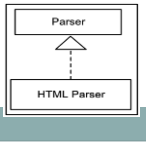
Modeling the Static Structure

Links and Relationships

- Inheritance/generalization: Generalization is the basic type of relationship used to define reusable elements in a class diagram. Child classes inherit the common functionality defined in the parent class.

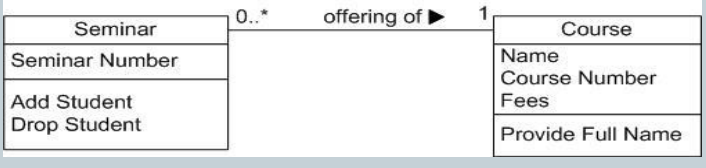


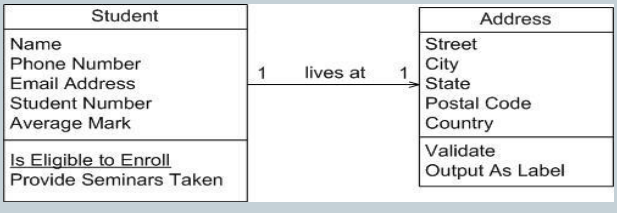
- Realization: In a realization relationship, one entity defines a set of functionalities as a contract and another entity realizes the contract by implementing the functionality defined in the contract.



21

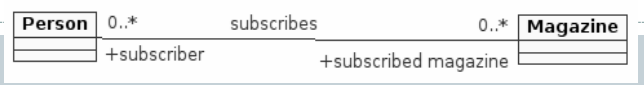
Association Relationship and Multiplicity





22

Association Relationships



Association

Directed Association

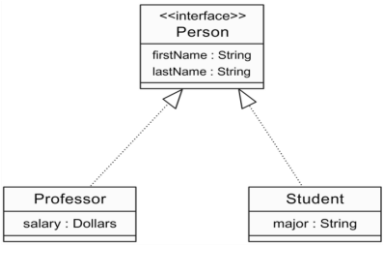
Aggregation

Composition

UML Class Diagram

Elements of a Class Diagram

- Interface: An interface is a variation of a class. While a class provides an encapsulated implementation of certain business functionalities of a system, an interface provides only a definition of the business functionalities of a system.

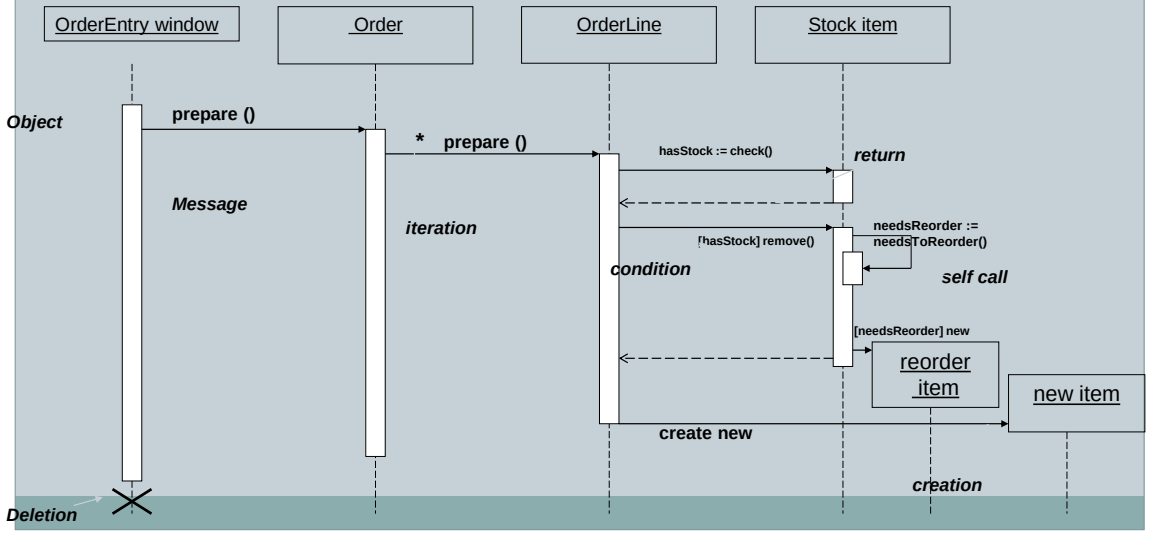


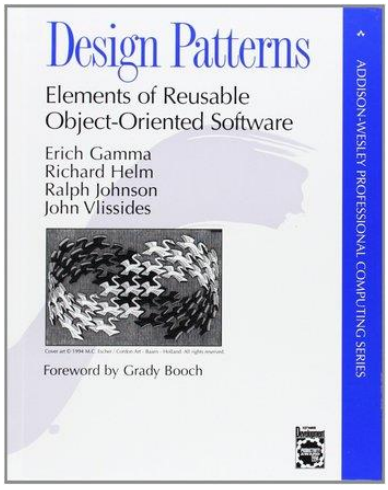
An interface is considered a specialization of a class-modeling element.

Sequence Diagrams

- Notations:
  - Rectangular box showing an object
  - Dashed line flowing from object known as lifeline
  - Activation box showing a method's lifetime
  - Arrows showing message/method
  - Half broken-head arrow describes an Asynchronous call
  - * showing iteration

Sequence Diagrams





GoF Design Patterns

27

List of GoF Patterns

Creational (5 + 2)	Structural (7)	Behavioral (11+1)
Factory Method	Adapter (object)	Interpreter
Abstract Factory	Bridge	Template Method
Prototype	Façade	Chain of Responsibility
Builder	Decorator	Command
Singleton	Composite	Memento
Monostate	Proxy	Iterator
Object Pool	Flyweight	Visitor
		Mediator
		Observer
		State
		Strategy
		Null Object Pattern

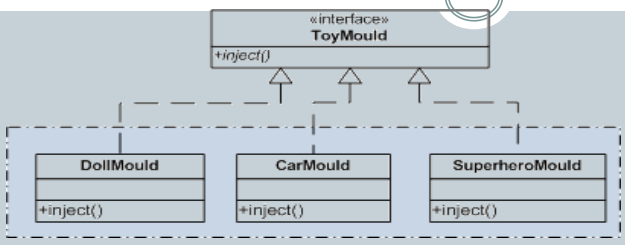
28

Creational Patterns

Concerned with the process of object creation.

Factory

Problem statement

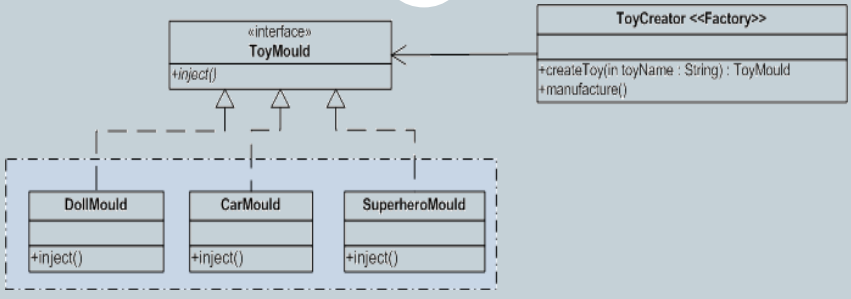



```
Class 1 →
If(toyName.equals("Ba
        toyMo
}
If(toyName.equals("
        toyMo
}
...
```

```
Class 2 →
If(toyName.equals("Barbie"){
        toyMould = new D
}
If(toyName.equals("Superman"
        toyMould = new S
}
...
```

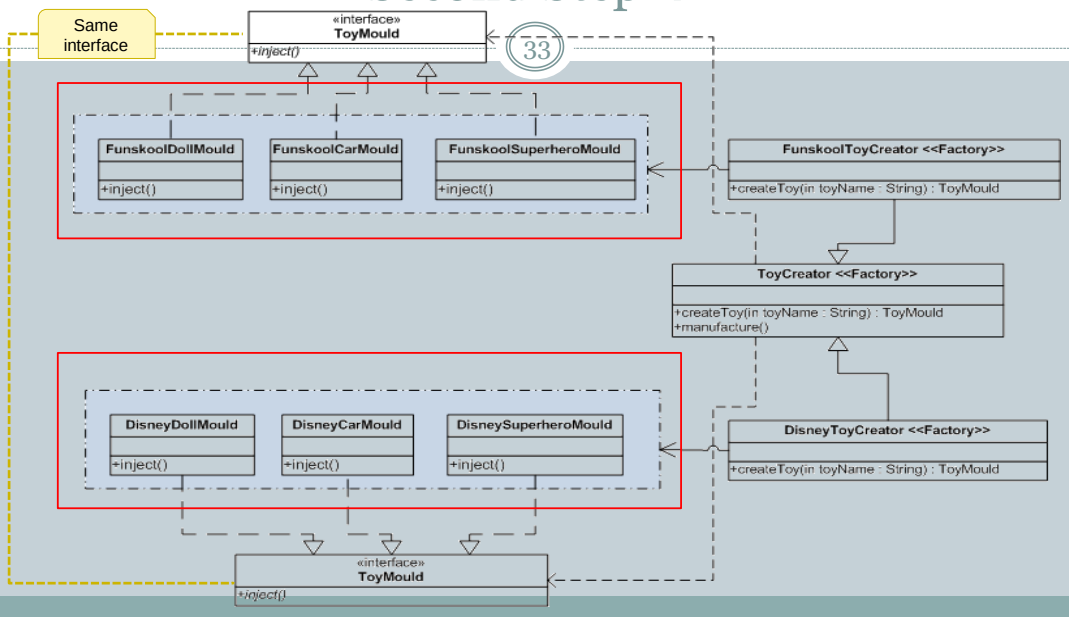
```
Class 3 →
If(toyName.equals("Barbie"){
        toyMould = new DollMould("Barbie");
}
If(toyName.equals("Superman"){
        toyMould = new SuperheroMould("Super man");
}
...
```


First step → Create a factory method



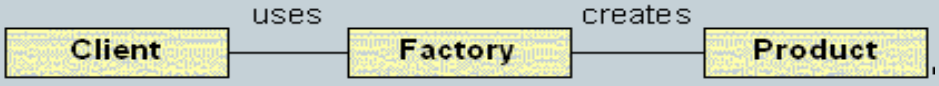

```
Toy Creator →
If(toyName.equals("Barbie"){
        toyMould = new DollMould("Barbie");
}
If(toyName.equals("Superman"){
        toyMould = new SuperheroMould("Super man");
}
...
```

Second Step →



What is Factory Method Pattern

- Methods, which create an instance of another class is known as a Factory Method.
- A class (Factory Class) can contain one or more Factory Methods. Factory Methods can be static or non-static



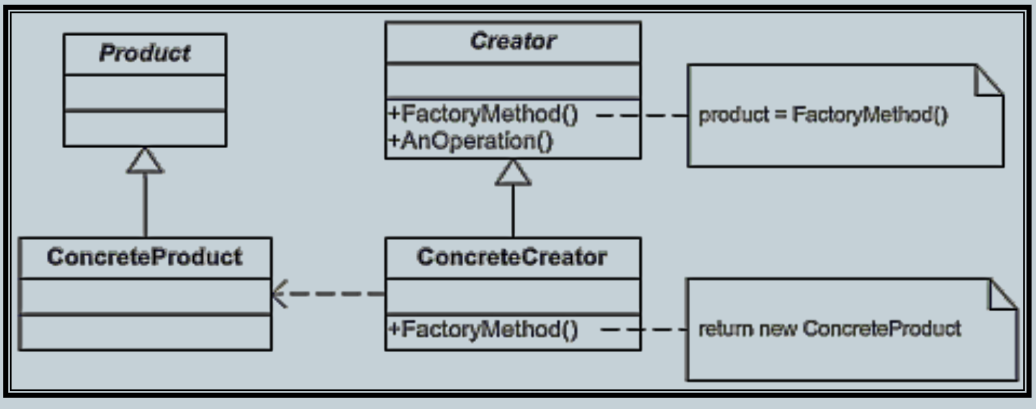
Implementation

35

- The client needs a product, but instead of creating it directly using the 'new' operator, it asks the factory object for a new product, providing the information about the type of object it needs.
- The factory instantiates a new concrete product and then returns to the client the newly created product(casted to abstract product class).
- The client uses the products as abstract products without being aware about their concrete implementation.

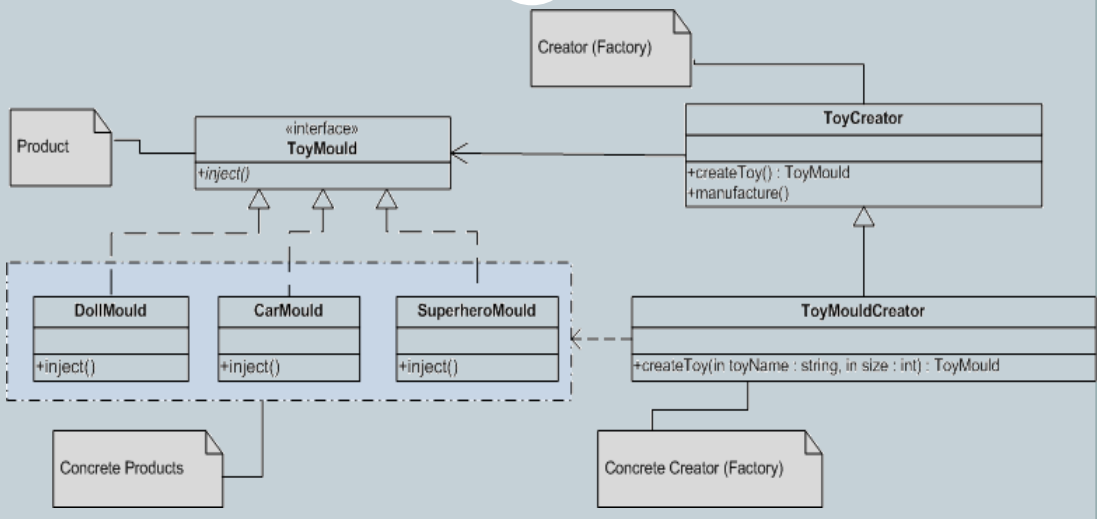
Factory Structure

36



Toys example (Structure)

37



Steps to create Factory method

38

- If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
- Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?
- Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.
- Consider making all constructors private or protected.

APIs using Factory

- Java
  - java.sql.DriverManager # getConnection(String url) {Returns a connection based on the url provided}
  - java.sql.DriverManager # getDriver(String url) {Returns Driver}
  - java.util.logging.Logger # getLogger(String name) { Finds or Creates a new logger for the given subsystem}
- .Net
  - System.Threading.Tasks.Task # TaskFactory.StartNew(Action<Object>, Object)

Consequences

- Adding a new concrete product or a family or concrete products only requires new corresponding concrete creators to be added. None of the existing creators get affected by the addition of a new concrete product.
- Addition of a new Abstract product results in all the existing creator classes getting affected. For example if we now consider including creating Games as a product to be added, we need to add a new abstract factory method viz, createGames and provide its implementation in all the controllers.
- Adding a new concrete product is easier than adding a new abstract product in the Factory pattern

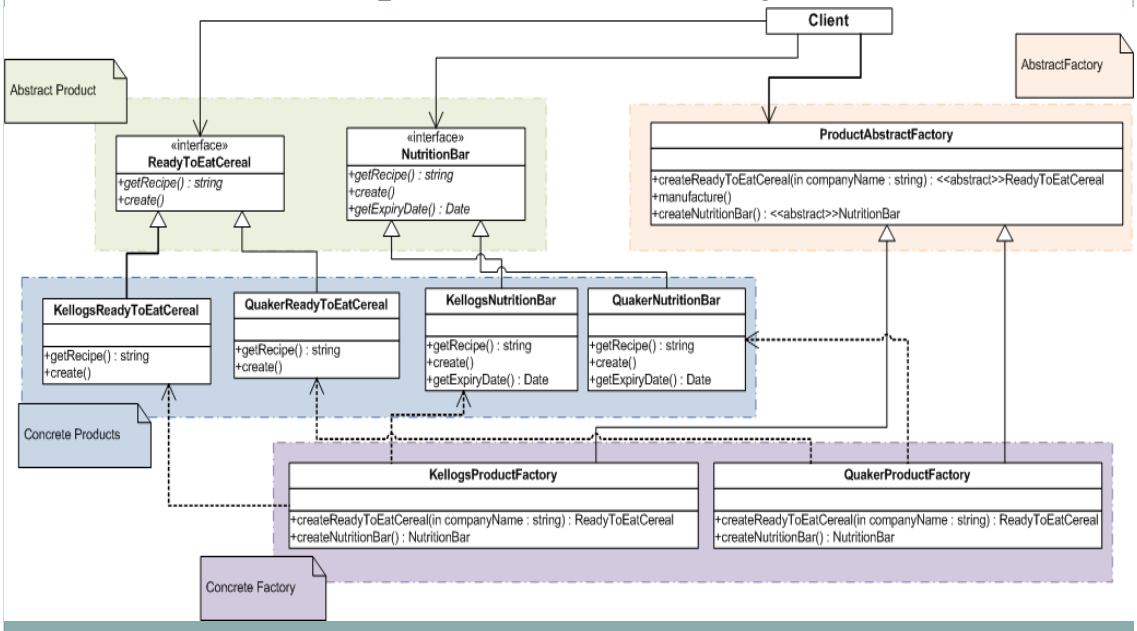
Abstract Factory

Abstract Factory

42

- Provides one level of interface higher than the factory pattern. It is used to return one of several factories
- Useful when there are multiple creators for the different products

Example Abstract Factory



Abstract Factory Pattern

Code for Abstract Factory

```
// Abstract Product A
abstract class NutritionBar{
    public abstract void getBarIngredients();
    public abstract void getNutritionValuePerServing();
}
// Abstract Product B
abstract class Cereal{
    public abstract void getCerealIngredients();

}
```

Family of Products

```
//ConcreteProductA
class KelloggsCereal extends Cereal{
    public void getCerealIngredients() { };
}

//ConcreteProductA
class QuakerCereal extends Cereal{
    public void getCerealIngredients() { };
}
//ConcreteProductB
class KelloggsNutritionBar extends NutritionBar{
    public void getBarIngredients() { };
    public void getNutritionValuePerServing() { };
}
//ConcreteProductB
class QuakerNutritionBar extends NutritionBar{
    public void getBarIngredients() { };
    public void getNutritionValuePerServing() { };
}
```

Concrete Products

Abstract Factory Pattern

Code Contd..

45

```
// Abstract Factory
abstract class CerealAbstractFactory{
    abstract Cereal createCereal();
    abstract NutritionBar createNutritionBar();
}
// Concrete Factory 1
class KelloggsConcreteFactory extends CerealAbstractFactory{
    Cereal createCereal(){
        return new KelloggsCereal();
    }
    NutritionBar createNutritionBar(){
        return new KelloggsNutritionBar();
    }
}
//Concrete Factory 2
class QuakerConcreteFactory extends CerealAbstractFactory{
    Cereal createCereal(){
        return new QuakerCereal();
    }
    NutritionBar createNutritionBar(){
        return new QuakerNutritionBar();
    }
}
```

Abstract Factory containing (factory) methods for family of products

Concrete Factory 1 containing (factory) methods for family of products

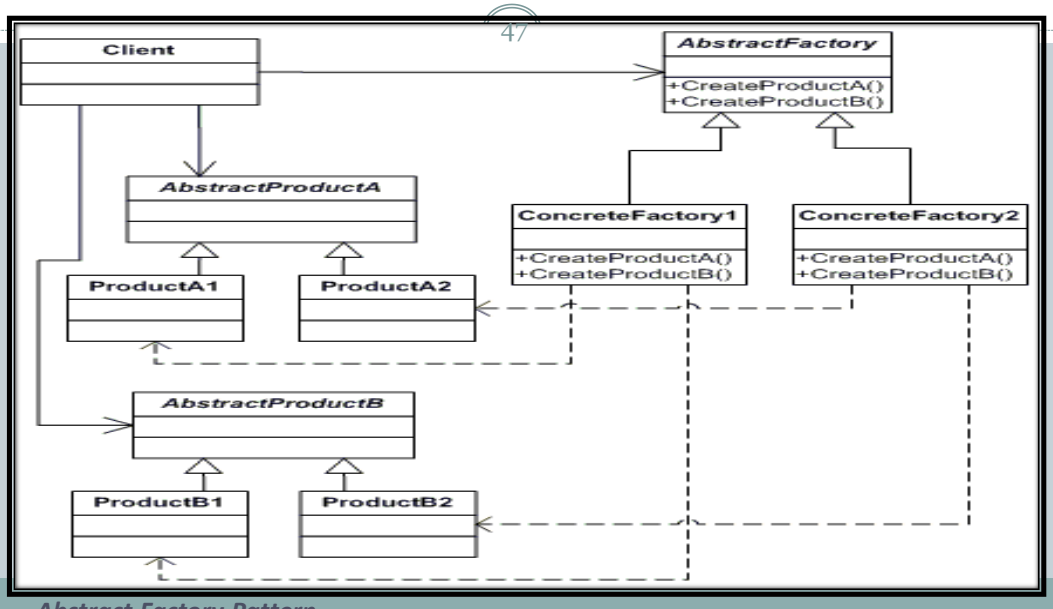
Concrete Factory 2 containing (factory) methods for family of products

Code contd..

```
package com.creational.abstractfactory;

//Factory creator - an indirect way of instantiating the factories
class CerealFactoryMaker{
    private static CerealAbstractFactory pf=null;
    static CerealAbstractFactory getFactory(String choice){
        if(choice.equals("Kelloggs")){
            pf = new KelloggsConcreteFactory();
        }else if(choice.equals("Quaker")){
                pf = new QuakerConcreteFactory();
            } return pf;
    }
}

//Client
public class TestCereal {

    public static void main(String[] args) {
        CerealAbstractFactory pf= CerealFactoryMaker.getFactory("Kelloggs");
        Cereal kellogsCereal =pf.createCereal();
        NutritionBar kellogsNB = pf.createNutritionBar();

        CerealAbstractFactory qpf= CerealFactoryMaker.getFactory("Quaker");
        Cereal quakerCereal =qpf.createCereal();
        NutritionBar quakerNB = qpf.createNutritionBar();
    }
}
```

Optional

Structure



Applicability

- You want the flexibility of choosing the class from an inheritance hierarchy of products to instantiate. There are multiple places across the system where the products need to be instantiated.
- As system needs to be configured with one of multiple families of products. A family of related products is meant to be used together, and you want to enforce this constraint.
- The concrete product classes have the same interface; they differ only in their implementations. You only want to reveal the interfaces to the clients using these objects.

APIs using Abstract factory

- java.sql.Connection # createStatement returns Statement
- java.sql.Connection # prepareStatement returns PreparedStatement
- java.sql.Connection # prepareCall returns CallableStatement

Checklist for choosing Abstract fact

- Decide if “platform independence” and creation services are the current source of pain.
- Map out a matrix of “platforms” versus “products”.
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the new operator.
- The client should retire all references to new, and use the factory methods to create the product objects.

Factory Method v/s Abstract Factory

Factory method returns an object based on a token:

```
RockFactory // Factory method
{
 Rock getRock( Rock.Type type );
}
```

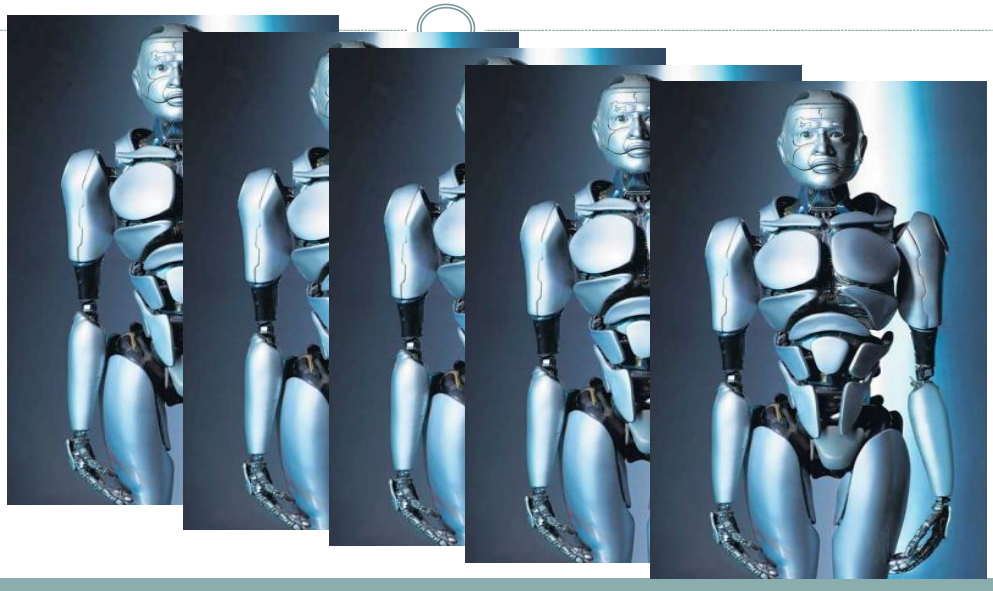
Abstract factory has a set of factory methods. You use it when you have a family of objects that you want to fetch from a single source. You create a factory based on a type and all of the individual objects are created based on the "container" type.

```
LandscapeFactory // Abstract Factory
{
 LandscapeFactory( LandscapeType type);
 Rock getRock();
 Bird getBird();
 House getHouse();
}
```

Abstract Factory Pattern

Prototype

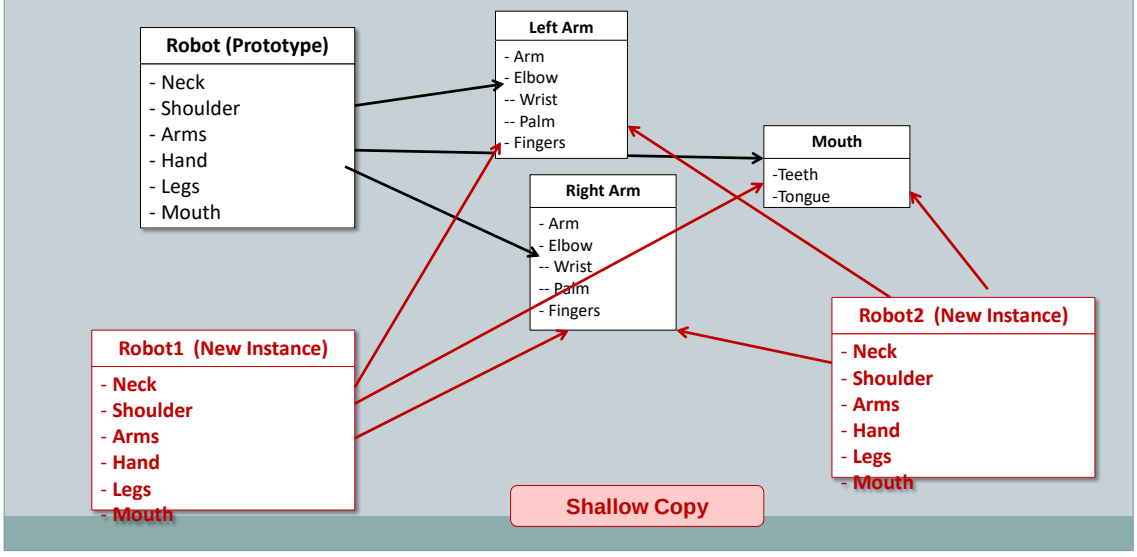
Robot Example



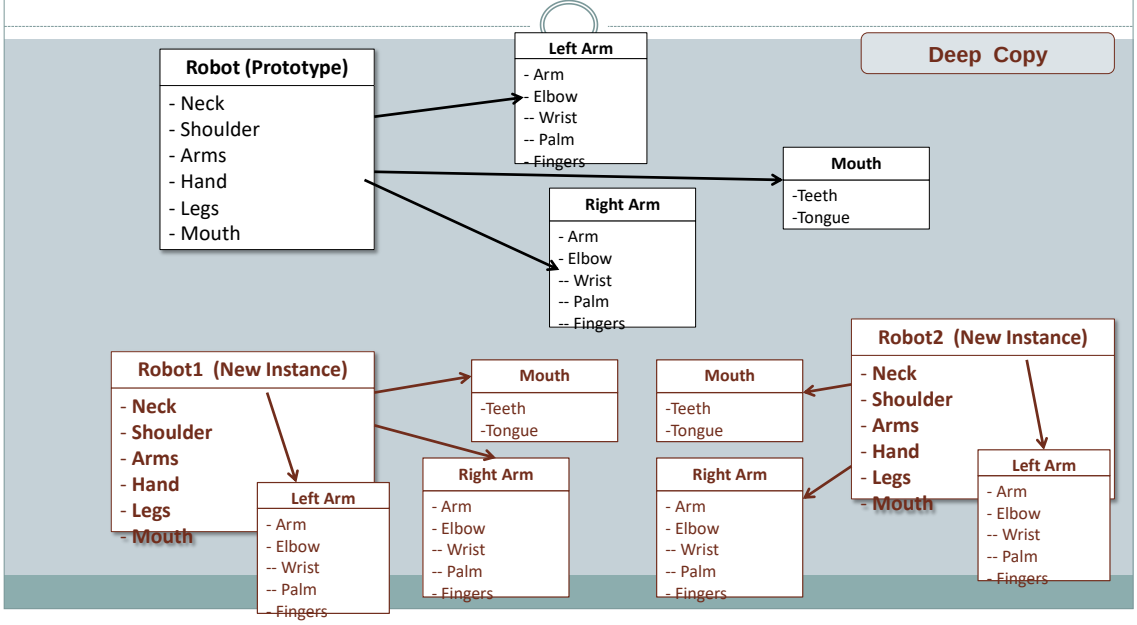
New Operation v/s Cloning

- The 'new' operator follows the following steps :
  A. Allocates a memory for the variable which will hold the reference of the object. Currently it will have null.
  B. The new operator then loads the class in the jvm
  C. It makes the call to the object constructor and executes the same
  D. If there are complex local variables being initialized in the constructor, it follows the above process for each type of variables.
- How to Clone in Java :
  - Implement Cloneable interface
  - Override clone() method provided by Object class and make it public
  - call super.clone() to create a clone (Memory to Memory Copy)

Shallow v/s Deep Cloning



Shallow v/s Deep Cloning



Prototype

57

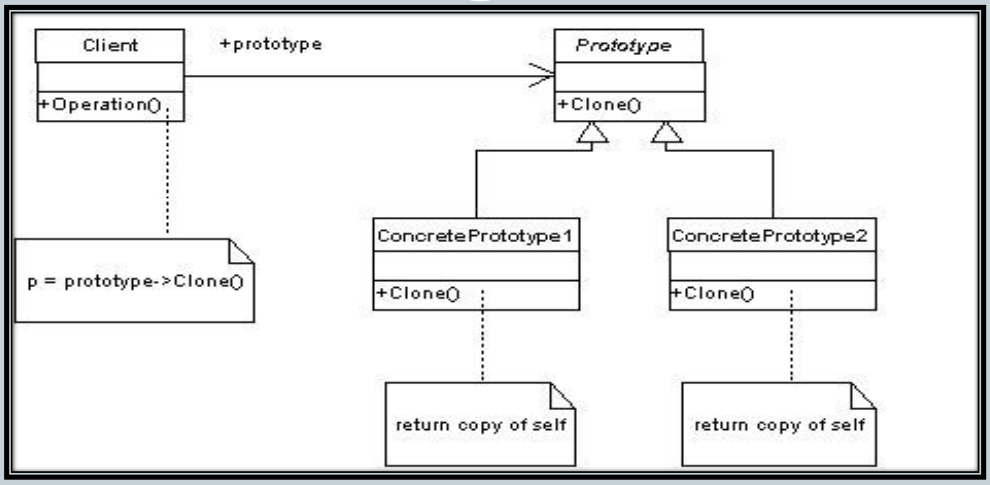
- Prototype means making a clone.
- If the cost of creating a new object is large and creation is resource intensive, we clone the object.

Intent

- Create new objects by copying this prototype object.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

Structure

58



Checklist for using Prototype pattern

59

- Add a clone() method to the existing "product" hierarchy.
- Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the "product" hierarchy.
- Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls clone() on that object, and returns the result.
- The client replaces all references to the new operator with calls to the factory method.

Prototype v/s Factory v/s Abstract Factory

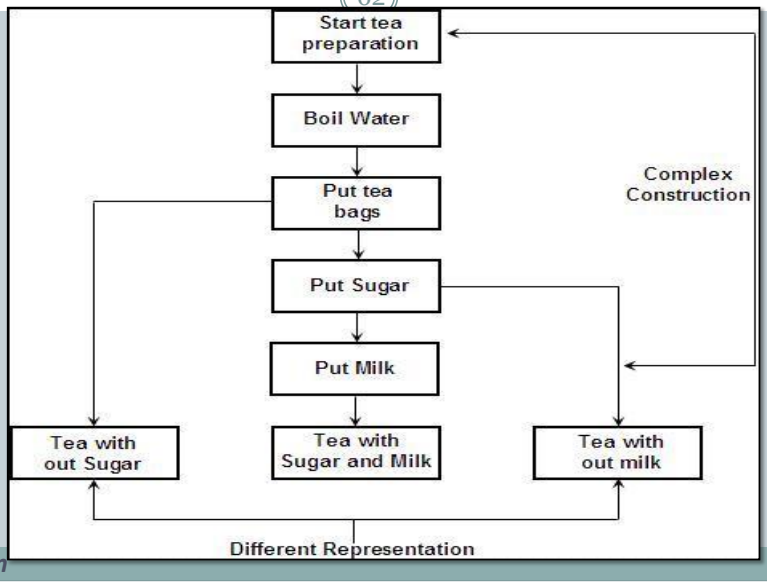
60

- Abstract Factory might store a set of Prototypes from which to clone and return product objects.
- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Prototype doesn't require sub-classing, but it does require an "initialize" operation. Factory Method requires sub-classing, but doesn't require Initialize.
- Prototype co-opts one instance of a class for use as a breeder of all future instances.
- Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive "creation from scratch", and support cheap cloning of a pre-initialized prototype.
- Prototype is unique among the other creational patterns in that it doesn't require a class – only an object.

Builder

Example : Creating Tea

62



Builder Pattern

Builder pattern

63

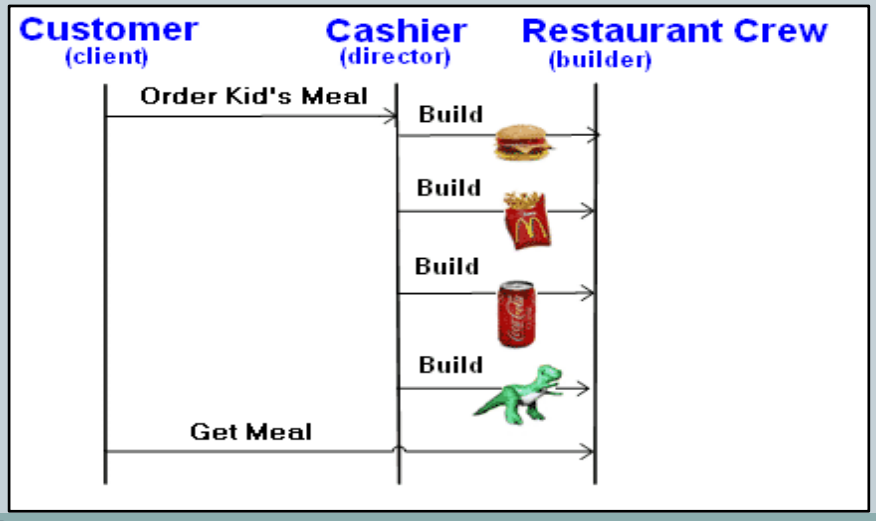
- Construct a complex object from simple objects step by step.

Builder Pattern is used when:

- the creation algorithm of a complex object is independent from the parts that actually compose the object
- the system needs to allow different representations for the objects that are being built

Example : Building a meal

64



Code for Builder pattern

```
// BUILDER interface
interface Item {

    /**
     * build is the method, as every item will be built and packed in a different wa
     * E.g.:- The burger will be packed as wrapped in a paper The cold drink
     * will be given in a glass The  fries will be packed in a card box
     * and The toy will be put in the bag straight. The class Packing is an
     * interface for different types of packing for different Items.
     */
    public Packing build();

     * price is the method as
    public int price();

}
```

```
//Product
interface Packing {

}
//Product
class Envelop implements Packing {

}
//Product
class Wrapper implements Packing {

}
//Product
class CoveredGlass implements Packing {

}
//Product
class MealBox extends ArrayList<Item> implements Packing {

}
```


Code contd..


```
// Crew who creates and packs fries into envelop
class FriesBuilder implements Item {
    public Packing build() {
        return new Envelop();
    }
    public int price() {
        return 25;
    }
}
//Crew who fills up Cola
class ColaBuilder implements Item {
    public Packing build() {
        return new CoveredGlass();
    }
    public int price() {
        return 20;
    }
}
```

```
//concrete builder
class DollBuilder implements Item {

    public Packing build() {
        return new CoveredGlass();
    }

    public int price() {
        return 5;
    }
}
```

```
//Restaurant crew who builds Burger
abstract class BurgerBuilder implements Item {

    public abstract int price();

}
// Restaurant crew who builds VegBurger
class VegBurgerBuilder extends BurgerBuilder {
    public Packing build() {
        return new Wrapper();
    }
    public int price() {
        return 39;
    }
}
```

Code Contd..

```
 * DIRECTOR
 */
class ChildMealDirector {

    public MealBox buildChildSpecialMeal() {
        MealBox mealBox = new MealBox();
        // Build the meal taking help from the builder objects
        Item[] items = { new VegBurgerBuilder(), new FriesBuilder(),
                new ColaBuilder(), new DollBuilder() };
        for (Item item : items) {
            mealBox.add(item);
        }
        return mealBox;
    }

    public static int calculatePrice(MealBox mealBox) {
        int totalPrice = 0;

        for (Item item : mealBox) {
            totalPrice += item.price();
        }
        return totalPrice;
    }
}
```

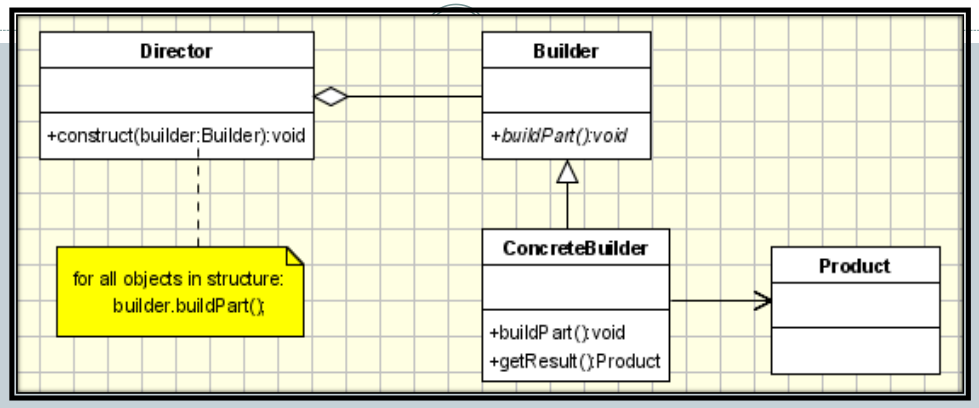
```
class Client
{
    public static void main(String[] args) {
        ChildMealDirector mealdirector = new ChildMealDirector();
        MealBox childMeal = mealdirector.buildChildSpecialMeal();
        System.out.println("Child meal price - "+
                mealdirector.calculatePrice(childMeal));
    }
}
```


Builder Pattern

68

- Builder pattern helps us to separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Builder pattern is useful when the construction of the object is very complex.
- If we are able to separate the construction and representation, we can then get many representations from the same construction.

Structure



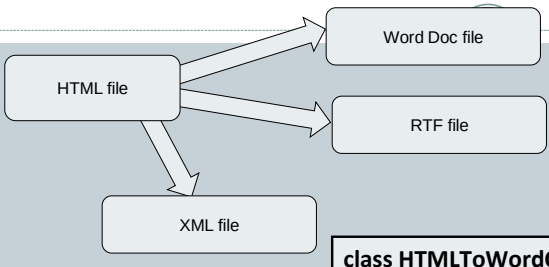
The **Builder** class specifies an abstract interface for creating parts of a Product object.
The **ConcreteBuilder** constructs and puts together parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for saving the product.
The **Director** class constructs the complex object using the Builder interface.
The **Product** represents the complex object that is being built.

How to use Builder Pattern

70

- Decide if a common input and many possible representations (or outputs) is the problem at hand.
- Encapsulate the parsing of the common input in a Director class.
- Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
- Define a Builder derived class for each target representation.
- The client creates a Director object and a Builder object, and registers the latter with the former.
- The client asks the Director to “construct”.
- The Director asks the Builder to return the result.

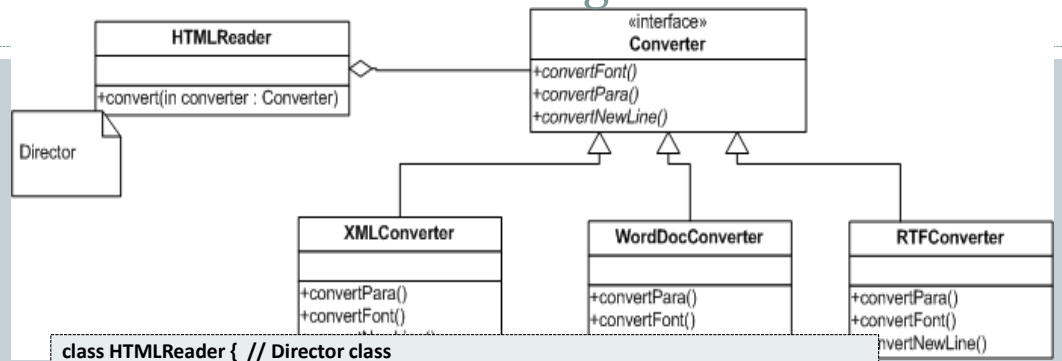
Another example – HTML Converter




```
class HTMLToXMLConverter {
    public void convert () {
        while (…) {
            if (tokenType = FONT_CHANGE)
                convertFont (font);
            if (tokenType == NEW_PARA)
                convertPara ();
            …
```

```
class HTMLToWordConverter {
    public void convert () {
        while (…) {
            if (tokenType = FONT_CHANGE)
                convertFont (font);
            if (tokenType == NEW_PARA)
                convertPara ();
            …
        }
    }
    …
}
```

HTML Converter using Builder Pattern




```
class HTMLReader {  // Director class
    public void convert (Converter converter) {
        while (…) {
            if (tokenType = FONT_CHANGE)
                converter.convertFont (font);
            if (tokenType == NEW_PARA)
                converter.convertPara ();
            …
        }
    } }
```

```
Client code
    HTMLReader reader = new HTMLReader("test.html");
    reader.convert(new WordConverter());
    reader.convert(new XMLConverter());
```

APIs and Builder

Java APIs

- java.lang.StringBuilder # append
- java.lang.StringBuffer # append
- @builder

.Net APIs

- System.Text.StringBuilder #Append

Singleton

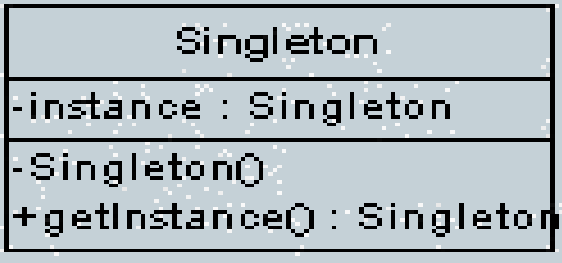
Singleton

75

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.
- Encapsulated “just-in-time initialization” or “initialization on first use”.

Structure

76



Implementing a Singleton

77

```
public class Logger {

    // private static reference of itself
    private static Logger loggerInstance;

    //private constructor
    private Logger(){
    }

    // method to return the single instance
    public static Logger getLogger()
    {
        if(loggerInstance == null)
        {
            loggerInstance = new Logger();
        }
        return loggerInstance;
    }

    //other methods
    public void logMsg(String priority, String message) {

    }
}
```

Singleton Pattern

Other implementations and their advantages/ disadvantages

```
// method to return the single instance
public static Logger getLogger()
{
    synchronized (loggerInstance) {
        if(loggerInstance == null)
        {
            loggerInstance = new Logger();
        }
        return loggerInstance;
    }

}
```

Using Synchronized

```
// Initialize the private static reference of itself
private static Logger loggerInstance = new Logger();

//private constructor
private Logger(){
}

// method to return the single instance
public static Logger getLogger()
{
    return loggerInstance;
}
```

Early Singleton : Initializing the logger instance during class load

Singleton Pattern

Benefits

79

- Controlled access to unique instance.
- Reduced name space.
- Allows refinement of operations and representations.
- Permits a variable number of instances.
- More flexible than class operations.

Monostate

```
public class Monostate
{
    private static string dataItem;
    public string DataItem
    {
        get { return dataItem; }
        set { dataItem = value; }
    }

    public Monostate(){ }
}
```

```
using System;
using static System.Console;

namespace DesignPatterns
{
  public class CEO
  {
    private static string name;
    private static int age;

    public string Name
    {
      get => name;
      set => name = value;
    }

    public int Age
    {
      get => age;
      set => age = value;
    }

    public override string ToString()
    {
      return $"{nameof(Name)}: {Name}, {nameof(Age)}: {Age}";
    }
  }

  static class Program
  {
    static void Main(string[] args)
    {
      var ceo = new CEO();
      ceo.Name = "Adam Smith";
      ceo.Age = 55;

      var ceo2 = new CEO();
    }
```

Points to discuss

81

- Singleton v/s MonoState
- Extending Singleton pattern for object pooling

Checklist to use Singleton patter

82

- Define a private static attribute in the "single instance" class.
- Define a public static accessor function in the class.
- Do "lazy initialization" (creation on first use) in the accessor function.
- Define all constructors to be protected or private.
- Clients may only use the accessor function to manipulate the Singleton.

Singleton

83

- **Benefits of the SINGLETON**
- Applicable to any class. You can change any class into a SINGLETON simply by making its constructors private and by adding the appropriate static functions and variable.
- Can be created through derivation. Given a class, you can create a subclass that is a SINGLETON.
- Lazy evaluation. If the SINGLETON is never used, it is never created.
- **Costs of the SINGLETON**
- Destruction is undefined. There is no good way to destroy or decommission a SINGLETON. If you add a decommission method that nulls out the instance, other modules in the system may still be holding a reference to the SINGLETON instance. Subsequent calls to Instance will cause another instance to be created, causing two concurrent instances to exist.
- Not inherited. A class derived from a SINGLETON is not a singleton. If it needs to be a SINGLETON, the static function, and variable need to be added to it.
- Efficiency. Each call to Instance invokes the if statement. For most of those calls, the if statement is useless.
- Nontransparent. Users of a SINGLETON know that they are using a SINGLETON because they must invoke the Instance method.

MonoState

84

- **Benefits of MONOSTATE**
- Transparency. Users of a MONOSTATE do not behave differently than users of a regular object. The users do not need to know that the object is MONOSTATE.
- Derivability. Derivatives of a MONOSTATE are MONOSTATES. Indeed, all the derivatives of a MONOSTATE are part of the same MONOSTATE. They all share the same static variables.
- Polymorphism. Since the methods of a MONOSTATE are not static, they can be overridden in a derivative. Thus different derivatives can offer different behavior over the same set of static variables.
- Well-defined creation and destruction. The variables of a MONOSTATE, being static, have well-defined creation and destruction times.
- **Costs of MONOSTATE**
- No conversion. A normal class cannot be converted into a MONOSTATE class through derivation.
- Efficiency. A MONOSTATE may go through many creations and destructions because it is a real object. These operations are often costly.
- Presence. The variables of a MONOSTATE take up space, even if the MONOSTATE is never used.

End of Chapter (Creational Design Patterns)